UA LIBRARIES
10024630 20

MANAGEMENT IMPLICATIONS OF MOVING FROM A TRADITIONAL STRUCTURED SYSTEMS DEVEOPMENT METHODOLOGY TO OBJECT-ORIENTATION

By:

Jinlan Tomasic

RECOMMENDED:

Advisory Committee Chair

APPROVED:

Dean, School Of Management

Dean of the Graduate School

1-7-03

Date

Management Implications of Moving From A Traditional Structured Systems Development Methodology to Object-Orientation

A

THESIS

Presented to the Faculty

of the University of Alaska Fairbanks

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF SCIENCE

By:

Jinlan Tomasic, B.S.

Fairbanks, Alaska

May 2003

QA
76.64
T657
2003

RASMUSON LIBRARY
UNIVERSITY OF ALASKA-FAIRBANKS

ABSTRACT

As software application systems become larger and more complex, many software employers and managers believe that the key to sustaining its competitive advantage in the computing technology market lies in its software engineering capabilities. Software crisis situation seems to be a common occurrence in the software development environment as systems become larger and more complex. Object Orientation (OO) has been proposed as a viable alternative to traditional approach (i.e., structured techniques), an approach that many hope will solve the current software crisis.

OO is a new paradigm, and it requires new types of knowledge, new specialists, and significant changes in the mindset, an entirely different way of thinking, representing and solving a problem. The transition of moving toward the OO from the traditional approach may involve a high risk of failure if the managers do not understand the nature of paradigm shifts and do not anticipate the future. The problem of moving to OO has become very important. An understanding of potential problems from migrating to the new paradigm helps managers make a smoother paradigm shift.

The implications and challenges of the OO paradigm are presented. The study suggests that Object-Oriented System Development (OOSD) requires more discipline, management and training than traditional software development does. Education and experience are keys for the success of any OOSD project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my committee chair, Dr. J. A. Lehman, for his help and guidance. His motivation and suggestions necessary to successfully complete a master thesis were greatly appreciated.

I would also like to thank Dr. D. O. Porter and Dr. J. H. Lee for their help and support as my advisory committee members.

Lastly, I would like to thank my husband, Leon, and my son, Tad, for their understanding of my long days of research and work. This thesis would not have been possible without their endurance, encouragement and support.

CHAPTER 1

INTRODUCTION

1.1 Software Development Problems

As software application systems become larger and more complex, many software employers and managers believe that the key to sustaining its competitive advantage in the computing technology market lies in its software engineering capabilities. Boehm defines software engineering as "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them" (Boehm, 1976). The notion of 'software engineering' was first proposed in 1968 at a conference held to discuss what was then called the 'software crisis' (Sommerville, 2001). Software crisis has been described in the situation such as projects delayed or not completed, cost overrun (i.e. higher than estimates), or delivered products not in good quality and failed to meet customer's requirements. This situation seems to be a common occurrence in the software development environment as systems become larger and more complex. In the 1970s, "Structured Programming" and "Structured Design and Analysis" were suggested to overcome "software crisis". However, software is still in crisis while the industry demands more complex application systems than ever. As Booch (1991) writes, "Our failure to master the complexity of software results in projects that are late, over budget, and deficient in their stated requirements". Such a crisis is often followed by the

emergence of a new paradigm, one that will make anomalies in the prior paradigm conform (Kuhn, 1970).

Object Orientation (OO) has been proposed as a viable alternative to traditional approach (i.e., structured techniques), an approach that many hope will solve the current software crisis. In addition to meeting the challenge of developing ever-more-complex software, OO also addresses two problems of structured techniques. The first is the rift between process and data and the mismatch between the requirements-analysis model and the software-design model. The second is the gap between so-called information and real-time system-development approaches (Page-Jones, 2000). This however does not mean OO is a miracle solution to all current software problems, neither is it a silver bullet for solving the problems besieging software industries. OO is a new paradigm, and it requires new types of knowledge, new specialists, and significant changes in the mindset, an entirely different way of thinking, representing and solving a problem. The transition of moving toward the OO from the traditional approach may involve a high risk of failure if the managers do not understand the nature of paradigm shifts and do not anticipate the future. Thus, an understanding of the software development life cycle (SDLC), the nature of the new software paradigm such as OO, the current development approach (e.g. structured approach), the difference between the paradigm change and the issues of migration to a new paradigm will help the manager to make a more smooth paradigm shift and ultimately succeed in the new paradigm.

1.2 Objectives of this Study

The primary objective of this study is to address the management implications and/or concerns in moving to the Object-Oriented system development from the traditional approach (i.e., structured approach.) Managers need to know more about the potential issues that could occur from the software paradigm change to make more informed decisions. Researchers have indicated that radical information technology (IT) innovations involve changes in knowledge, information, values, power, incentives and commitments. The extent to which an innovation is considered radical depends on the degree to which it differs from the current environment. As Orlikowski writes, "radical change implies a paradigm shift, which requires a reframing and renegotiation of the IT mission, role, and relationships in the organization" (Orlikowski, 1993). An understanding of exact nature of OO, the traditional (i.e. structured) approach and the difference between them is important for the managers to make a smooth paradigm transition. This study also shows the basic concepts of the OO and structured technologies, the difference between them along with the advantages and disadvantages of the paradigm change. An overview of the software development life cycle (SDLC) is also provided.

To summarize, the objectives of this study include the following:

- Overview the Software Development Life Cycle (SDLC)
- Identify the Software Development Life Cycle (SDLC) Methodologies
- Show the OO and Structured Approaches in Software Development
- Describe the Advantages and Disadvantages of Structured Approach

- Describe the Advantages and Disadvantages of OO Approach
- Compare the OO Paradigm from the Structured Approach
- Explore the Management Implications in Moving to a New Paradigm
- Suggest Appropriate Actions in Managing New Paradigm Issues
- Conclude this Study and Provide Directions for Future Anticipation

1.3 Scope and Limitations of this Thesis/Research/Study

Page-Jones' Fundamentals of Object-Oriented Design in UML and Yourdon's Object-Oriented Systems Design are the conceptual basis for this thesis. In the context of this study, software development refers to analysis, design, and programming. Structured analysis, design and programming contain the traditional approach to software development while the new paradigm refers to object-oriented analysis (OOA), object-oriented design (OOD), and object-oriented programming (OOP). This study attempts to show the differences between the traditional structured approaches to software development and OO, and the management implications of moving to the OO paradigm from structured approach. It must be emphasized that the approaches mentioned here are for research in software development and not for the entire software engineering discipline. Therefore, details of the SDLC, testing, maintenance, documentation and process improvement maturity model, etc. are beyond the scope of this study.

1.4 Organization of this Study

Chapter 2 presents a historical perspective of software system development, followed by a detailed comparison of the traditional approach (i.e. structured approach) with object-oriented system development. The concept along with the advantages and

disadvantages of the paradigms are also presented. Chapter 3 begins with a review of the problem and is followed by detailed description of potential issues of moving to the OO paradigm. Suggestions are presented at the end of chapter. Chapter 4 presents a summary and directions for future study.

CHAPTER 2

SOFTWARE SYSTEMS DEVELOPMENT

2.1 Historical Background of Software Systems Development

The beginnings of the software system development approaches can be traced to Dijkstra, who supposed that any arbitrary program could be proved to be mathematically correct or incorrect (Berard, 1993). In 1968, the well-known computer scientist E.W. Dijkstra shocked the programming world in response to the abuse of a "GO TO "statement with his letter titled "Go To Statement Considered Harmful" to the Communications of the Association for Computing Machinery (ACM) (Dijkstra, 1968). The results of his efforts have provided the basis for the structured programming. During the late 1960s, researchers and authors, such as Larry Constantine, realized that structured programming alone was not enough to develop robust systems and that there was a need to identify potential problem areas before implementing the program. The pioneering work of Constantine and his students Glen Myers and Wayne Stevens on "Structured Design" was published in IBM Systems Journal (Stevens, Myers and Constantine, 1974). In the mid to late 1970s, there were a number of significant contributions to the structured system development. For examples, "Structured Design – Fundamentals of a Discipline of Computer Programs and Systems Design" co-authored by Yourdon and Constantine (Yourdon and Constantine, 1979); and DeMarco's "Structured Analysis and System Specification" (Demarco, 1978). In the 1980s, the

structured system development evolved on structured analysis, design and programming, and the focuses were leaned toward the prototyping approaches (e.g., Boehm, 1988), the real time issues (e.g., Ward and Mellor, 1985) and to the Computer Added Software Engineering (CASE).

The Object-Oriented Programming (OOP) concepts have been around since the 1960s. Simula, the first OO programming language, was introduced in 1967 (Kirkerud, 1989). The central ideas of Simula were used as a basis to develop Smalltalk, which is considered by many to be an archetypal object-oriented programming language (Berard, 1993). Both Simula and Smalltalk are regarded by many as exemplars of true object-oriented programming languages. It was not until the 1980s that object-oriented analysis (OOA) and object-oriented design (OOD) started attracting attention.

2.2 Software Development Life Cycle (SDLC)

The field of software engineering began to emerge in the 1960s to provide a scientific approach to software development. It was believed that such an approach would enable software developers to cope with the inherent complexity of large systems (Sommerville, 2001). The basic software engineering philosophy is to use process and technology that support an incremental and evolutionary approach to delivering quality solutions. This philosophy provides one of the best ways of managing risk in an environment where there is incomplete and changing understanding of the problem to be solved and the technology available to solve it.

According to IEEE for Software Engineering (IEEE, 1991), Software Development Lifecycle (SDLC) is defined as the period of time that begins with the

decision to develop a software and ends when the software is delivered to its end user. The software development life cycle typically include the following processes (Pressman, 2001):

- System engineering and analysis

  Since software implementation is usually part of a larger system, work must begin by establishing requirements for all system components and subsets. A system review and analysis is essential when software must interface with other elements such as hardware, people, database and client/server environment.

- Software requirements analysis

  The requirements gathering process is critical and focused on the software. Sometimes the requirements could be part of the solutions to a problem, and these need to be identified and used for verification and validation (V&V) later. In order to have a successful implementation, the developer must understand the information domain for the software, as well as the required functions, performance and interfacing. Requirements and/or problem resolutions for both the system and the software must be documented and reviewed with the clients.

- Design

  The design process translates requirements into a representation of the software (e.g. pseudo code) that can be reviewed for quality before coding begins.

- Coding

  Coding is to translate the design into a machine-readable form.

- Testing

Once code has been generated, testing begins. The testing processes include: white box, black box, integration, and full system testing.

- Maintenance

Changes are part of the SDLC. Software maintenance reapplies the each of the above steps to an existing program rather than a new one. The life-cycle steps described above are known as the classic life cycle for software engineering.

2.3 Software Development Life Cycle (SDLC) Methodologies

Software engineering is comprised of a set of steps that are used for the improvement of the quality of the development/maintenance work throughout the software development life cycle (SDLC). These steps such as the classic life cycle are often referred to as software engineering paradigms (models.) Based on the type of the paradigm, the process steps vary. There are various models used by different authors of the software engineering. According to Pressman (Pressman, 2001), the following models are widely discussed (and debated):

- The Classic Life Cycle (sometimes called the "waterfall model")

  The classic life cycle is the oldest and the most widely used model for software engineering. This model uses a systematic, sequential approach to software development that begins at the system level and through the requirement analysis, design, coding, testing, and maintenance. Due to the lack of iterations, this model has received some criticism over the past decade. Disregarding the problem(s), this model has, however, been used as a template for the conventional process steps such as analysis, design, coding testing, and maintenance.

RASMUSON LIBRARY
UNIVERSITY OF ALASKA-FAIRBANKS

- Prototyping

Prototyping is a process that enables the developer to create a model of the software that must be built. Prototyping begins with requirements gathering, then a "quick design" which leads to the construction of a prototype, and then the prototype is evaluated by the customer/user and is used to refine requirements for the software to be developed. The prototyping can be problematic for the developer to rush into a working product without considering the overall software quality or long-term maintainability. Although problems can occur, prototyping is an effective model for a quick working model.

- The Spiral Model

The spiral model was developed to enhance both the classic life cycle and prototyping through iterations while adding a new process – risk analysis. This model encompasses the following four major activities (Pressman, 2001):

1. Planning – determination of objectives, alternatives and constraints
2. Risk analysis – analysis of alternatives and identification/resolution of risks
3. Engineering – development of the "next-level" product
4. Customer evaluation – assessment of the results of engineering

With each iteration around the spiral model, progressively more complete versions of the product are built. This model uses prototyping as a risk reduction mechanism while maintaining the systematic stepwise approach suggested in the classic life cycle, and incorporates it into a iterative framework to be more realistic in the real world. It allows the developer and customer work closely to

understand the system requirements and react to the risks at all levels. The Spiral model is currently the most realistic and popular approach to the development for large scale systems.

In his "Software Engineering" book 6th edition, Sommerville suggested the following process model for some software development (Sommerville, 2001):

- Reuse-based development model

  This model is based on the existing reusable components in the system. The development process focuses on integrating these components into a system rather than developing them from scratch. The generic process model for the reuse-oriented development includes: Requirement specifications, Component analysis, Requirements modification, System design with reuse, Development and integration and System validation.

The reuse-oriented model has the obvious advantage that it reduces the amount of software to be developed and so reduces cost and risks. It usually leads to fast delivery of the system. However, this may lead to a system which does not meet the real needs of users due to requirements compromises. Also, some control over the system evolution may be lost as new versions of the reusable components are not under the control of the original system.

2.4 Two Approaches to Software Systems Development

For the purpose of this study, there are two basic approaches to software systems development. These two options are: (1) Traditional (i.e. Structured) System Development techniques, which comprise two separate Process-Driven and Data-driven

architectures. (2) OO Development approach which models software systems as a collection of objects that interact with each other. These two approaches are summarized in the following sections:

2.4.1 Traditional (Structured) System Development

Structured techniques have evolved from structured programming through structured design to structured analysis (Nerur, 1995). Structured analysis is the study of the problem domain in order to define the requirements for solving the problem. The emphasis is on what the system is intended to do, and is therefore considered as the business function part of systems development. Structured analysis is followed by a detailed structured design which views the system in terms of how it will accomplish its process-oriented goals. The term - structured design was introduced by IBM in an article in the IBM system Journal in 1974 (Stevens, Myers, and Constantine). Prior to that article, the various concepts were referred to as modular design, logical design, composite design, or the design of program structure. Structure design concerns the architecture, organization, and structure of computer program and of systems of programs. Both structured analysis and structured design put strong emphasize on data flow. Structured specification is expressed by means of the tools of Structured Analysis: Data Flow Diagrams (DFDs), Entity Relationship Diagrams (ERDs), data dictionary, structured English, decision trees, decision tables, and data access diagrams. Structured Design uses two additional tools: pseudo code and structure chart (Page-Jones, 1988).

Structural System Development can be easily implemented with procedural programming languages. Languages that use procedural abstractions are called

imperative languages (Schneider and Gersting, 1999). Software systems built with this approach tend to be difficult to extend, modify, and maintain

2.4.2 Object-Oriented Approach

The OO approach emphasizes the encapsulation of data and procedures. The only items of interest in analysis, design, and implementation are objects (Korson and McGregor, 1990). The interface and behaviors of an object are determined by its collections of operations or methods. The messages to which an object responds depend on these methods (Wegner, 1990). Therefore, an object has state, behavior and identity.

The term 'object' was first used in the software world in the SIMULA programming language, to simulate some aspects of reality – it means a combination of data and logic that represents some real world entity. UML Semantics Guide defines that an Object is an entity with well-defined boundary and identity that encapsulates both state and behavior (OMG, 1999):

State is represented by attributes and relationships;

Behavior is represented by operations, methods and state machines.

A class is a description of a set of objects that share the same attributes, operations, methods, relationships and semantics (OMG, 1999). Thus, an object is an instance of a class. For example, a group of personal vehicles can be defined as a class of car; a specific car (e.g., my car) is an object and an instance of the class of car. The "current state" is determined by the values of the properties or attributes. The values of the properties or attributes of the car include, for instance, RegisterNumber, Model, Type,

Color, and Speed. The "behavior" of the car may include Start, Off or ChangeSpeed, or getWinterized(?) - where '?' is a variable parameter.

Page-Jones (2000) clarifies the distinction between a class and an object as follows:

- A class is what you design and program
- Objects are what you create (from a class) at run-time.

This distinction clearly describes the structure of an object from its class in the OO programming and development. The implementation of the behavior of an object is internal. The services of an object may be obtained only through its interface. The interfaces between two objects are well defined and its implementation details are hidden from other objects. Objects that require the services of another object do not have to know anything about how its behavior has been implemented. This adheres strictly to the principle of information hiding. Objects that share a common structure and a common behavior are grouped into a class. A class whose instances themselves are classes is called metaclass (Berard, 1993). There is an inheritance relation between classes that establishes specializations and generalizations of concepts represented by classes (Korson and McGregor, 1990). This allows a subclass to inherit from its superclass, and also permits a new class to be created by changing an existing class. Pressman (2001) defines superclass as a collection of objects, subclass as an instance of a class and class hierarchy as attributes and methods of a superclass that are inherited by its subclasses. The following salient features are excerpted from Pessman's "Software Engineering: A Practitioner's Approach" to further illustrate the principle of the OO paradigm:

> Inheritance is like a template for attributes and operations. The template is overlaid on an object definition, enabling the object to use all attributes and operations defined for its class. Encapsulation is a packaging mechanism—it allows the analysis to combine data and process and refer to them with the same name. Polymorphism allows us to use the same method name to refer to operations of different objects. For example, we might use DRAW as a method for drawing different instances of the class SHAPE: circle, rectangle, triangle, etc (Pressman, 2001).

Another useful definition of the OO concept is Dynamic Binding. Booch (1991) defines dynamic binding as "Binding denotes the association of a name (such as a variable declaration) with a class; dynamic binding is a binding in which the name/class association is not made until the object designated by the name is created (at execution time)" (p.513). More useful discussions on the OO concepts are presented in Booch (1991), Berard(1993), Korson and McGregor (1990), Page-Jones(2000) and Pressman (2001).

In the OO approach, the architecture of a system is expressed as a collection of interacting objects that collaborate with one another to meet its goals and objectives. The communication between objects is handled through messages. Messages are the means by which objects exchange information with one another (Pressman, 2001). A complex system can be understood as a hierarchy of classes and objects. Many authors believe that OO is a natural way of dealing with complex systems. Object-Oriented consultants and outside advisors are likely to recommend a prototyping or spiral life-cycle model due to the general approach of the data and process to developing systems. The change from a waterfall conventional SDLC to a prototyping SDLC is a substantial mind-set change.

OO systems are almost always associated with Graphic Users Interface (GUI) while the older structural methodologies, which followed mostly a waterfall life cycle, were originally associated with character-based user interfaces. GUI projects are best developed by using prototyping paradigm; therefore, OO development follows closely the (fast) prototype life cycle. Some organizations, however, adopt a hybrid approach that employs OO technology for requirements analysis and software design, but use a traditional structured language (e.g. 'C' programming language), for implementation. For example, Foliage Software Systems uses this hybrid approach to deliver their user's interface aviation application systems to their clients. As Page-Jones noted that many so-called object-oriented systems are actually hybrids of object-oriented code and standard procedural code (Page-Jones, 1995). The OO development approach has also become very popular in client-server network environments.

2.4.3 Pros and Cons of Structured Approach

An understanding of the advantages, disadvantages and issues of the structured approach will help managers make a sound decision in supporting the migration of the new paradigm. The following discussions and the advantages/disadvantages of the OO in the next section will form the basis for the presentation of the OO research in next chapter.

Advantages

The main advantages of structured approach are described in the following:

- Software development is seen as a specification-driven process. An abstract specification is used as a basis for development procedures in the problem resolution.
- The software development process is a consistent series of steps. The waterfall model is the dominant basis for the process and this is characterized by a uniform and orderly sequence of steps.
- Every requirement of the software development can be traced back and verified in the specifications of analysis. Every requirement specified in the structured analysis step is used as a basis for further development in the design, coding and testing processes of the SDLC; therefore, it is traceable and verifiable.

Disadvantages

Before the introduction of Object Oriented Analysis and Design, most Information Systems professionals were taught that the structured approach was the proper way to approach software development. But they are coming to realize that these methods have shortcomings that include the following:

- The structured approach does not recognize user's need for changes. Users don't usually see their requested product until after the program is delivered. The traditional waterfall approach to software development dictates the completion of the analysis phase before design and likewise the completion of all design steps before construction. This assumption has an inherent risk of carrying forward incorrect or incomplete analysis into design and construction (Pei, 1995).

- Reusability is not encouraged (Booch, 1991). The software procedures and functions developed under the structured approach usually have specifications for solving particular problem domains thus cannot be easily reused without modifications. Any changes to the software could have undesirable side effects.
- There is no unifying model to integrate the SDLC phases from analysis, design, to implementation. It is difficult for a team to work on the same project in moving from one phase to another throughout the SDLC.
- Concurrency is not encouraged. Decision making and subroutine calls are done sequentially in the main program, i.e., it is centralized (Zeigler, 1990).

Management Issues

The following information regarding management issues of the structured approach was adopted from Yourdon's Managing the Structured Techniques (Yourdon, 1986):

Some arguments for abandoning the structured approach are, for example, as following:

- Problem with older Methodologies
- Radically different types of systems being developed

Technical concerns:

1. Limitations in computer hardware technology and systems software made adopting the structured techniques difficult.
2. Constraints in new technologies such as automated tools that make it possible to develop and maintain the graphical models of structured design efficiently.

Management concerns: The introduction of structured techniques usually introduces the following management problems:

1. The introduction of top-down implementation requires a change in the sequence in which programmers test their code and a change in the manner in which the system is delivered to the customer.
2. The introduction of chief programmer teams can be misinterpreted for job security (e.g. replacing current staff of "average" programmers with a few knowledgeable programmers who are familiar with the structured techniques).
3. Introduce a whole new method of communication between the end-user and the systems analyst.

2.4.4 Pros and Cons of OO Approach

Although many well-known software development practitioners believe OO approach has distinct advantages over traditional systems developments, others disagree with such opinion. This section provides detailed advantages, disadvantages and some management concerns to help managers in understanding the effect of the paradigm shift and to better anticipate the future.

Advantages

The main advantages of OO are that it:

- Encourages reusability;
- Is flexible and more resilient to change. Berard (1993) says that it can lead to systems that are easily modified, extended and maintained;

- Focuses on understanding of the problem domain (Coad and Yourdon, 1990). Solutions closely resemble the original problem (Berard, 1993);
- Reduces development risk (Booch, 1991);
- Leads to more stable, robust systems (Coad and Yourdon, 1990);
- Is intuitively appealing and closer to the real world problems;
- Reduces size of source code and shortens development time (Booch, 1991).

The details of these advantages of the OO approach are discussed in the following sections. The information described here regarding the advantages of OO approach over traditional structured approaches is adopted from Mohanarajah's Object Oriented Analysis and Design (Mohanarajah, 2002):

Advantages Due to Object Level Abstraction,

- Software (SW) people's productivity is escalated.
- Rework due to misunderstanding is much reduced - The solution model resembles the problem model. So, the problem domain is well understood by all the people involved in the development. And it is easier for communication between the client, user and software people.
- Increase Reusability - The whole object may be used in the same project or in other projects with subtle changes. A part of analysis or design portions also may be used again (e.g., analysis/design patterns and frameworks).
- Easy to adopt, extend and maintain - An entire object may be replaced, added or deleted, or a method of an object may be changed with negligible effects on other parts.

- Increase reliability - Some of the objects, building blocks of the software, might have been used reliably in other projects (already tested effectively).
- Decrease testing time - One reason for this is using reusable components, and this saves the time in testing phase.

Advantages Due to Seamless Transition,

- Decrease development time - Transition process is easy, so it takes less time since the concept is common and complexity is low. Hence the whole development time is reduced
- Reduce the size of the project. - Redundancy is avoided because it need not to repeat the same thing in different programs.
- Increase reliability - Errors introduced during transition process are almost nil.

These advantages of OO approach certainly re-enforce the positive view points in the management's support of moving to the new paradigm from the traditional approach.

The following is a summary used often for arguments in favor of OO (Yourdon, 1994):

- Increased productivity (due to reuse)
- Rapid Systems Development (due to reuse)
- Increased Quality (i.e. fewer defects from reuse & encapsulation)
- Increased Maintainability
- Radically different programming languages, development environments, and CASE tools

For the OO enthusiasts, it is hoped that OO technology will succeed in the software industry and be likely to follow the same trend as seen for other new technologies.

Disadvantages

It is important to note that no one method, technology or technique will solve all the problems associated with reuse. There will always be complications and/or disadvantages of moving to the new paradigm, and these must be expected and planned for. Some claimed disadvantages of the OO approach are as follows:

- Decreased system run-time performance
- Unavailability of adequate OO Database Management System (DBMS)
- Increased initial development time
- Unavailability of OO CASE tools
- Confusion from too many OOA/D methods
- Inability to try Object-Oriented System Development (OOSD) before committing
- Complexity of OOA/D methods
- Complexity of OOP languages
- Difficulty learning OOA/D methods
- Emphasizes on objects brings an emphasis on static modeling. Not clear that the modeling primitives are appropriate (Mohanarajah, 2002).
- Strong temptation to do design rather than problem analysis (Mohanarajah, 2002).

An Internet survey conducted by the University of Missouri - St.Louis (UML, 2000) found that both OO and non-OO developers view OOSD as superior, and OO developer hold this view more strongly than non-OO developer, and all developers view the reported disadvantages of OOSD as non-existent.

Management Issues

The steadily growing popularity of OO technology does not mean that OO is the highest priority for some software development organizations attempting to improve its productivity and quality. The following survey conducted by Systems Development, Inc., in 1991, documented the reasons why the non Object-oriented organizations chose not to use OO technology (Survey of Advanced Technology, 1991):

Table 1: Reasons for Not Using OO Technology

Reasons for Not Using	**% of Total**
Not aware of technology	31.0%
Benefits not demonstrated	3.5
No business need	17.2
Technology too costly	0.9
Organization unprepared	19.8
Technology too immature	19.8
Other	7.8

Additional arguments for not using the OO techniques are as follows (Yourdon, 1994):

- The lack of preparation – may involve training, familiarity with new concepts or a more fundamental problem involving a formal, standardized software development process
- The technology is too immature
- The lack of OO CASE tools
- The lack of language
- The lack of class libraries or other components of technology

A conclusion drawn from the survey of 150 randomly selected systems developers by the University of Missouri - St.Louis (2000) revealed that OOSD is alive and well. Experienced OOSD developers hold very strong beliefs to nearly all the OOSD advantages, and didn't believe in most reported disadvantages of OOSD. Although non-OO developers have some concerns about some aspects of OOSD, such as performance and development time, they didn't believe in many of the reported disadvantages of OOSD. Overall, this survey concludes that the hype surrounding OOSD looks more like reality and that adopting OOSD may indeed be worth the related time, effort, and cost (UMSL, 2000).

2.5 Comparison of Structured Paradigm and Object-Oriented Paradigm

2.5.1 Structured Versus Object-Oriented Paradigm

Both structured approach and Object-Oriented Paradigm are focused on system analysis and design using tools. The main difference between structured approach and OO are as follows:

- While structured approach emphasizes functions, OO emphasizes objects. Structured Analysis and Structured Design (SASD) decomposes the system into single-functioned modules as implementation methods. Object-oriented analysis and design (OOAD) revolves around objects, which themselves may have many attributes and functions.
- OOAD avoids the fragmentary nature of structured analysis. However, the objects may be inappropriate for system modeling and may raise the temptation to do design rather than problem analysis (Pfeiffer and Zhang, 2002).
- OOAD's emphasis on objects are prone to static modeling. As a contrast SASD is especially good for real-time system (Pfeiffer and Zhang, 2002).

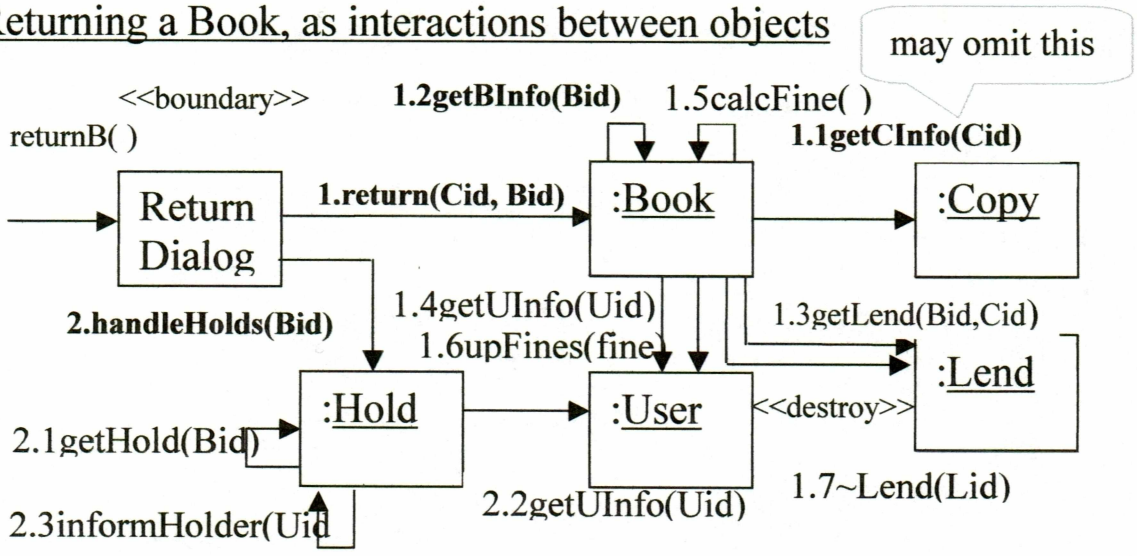
The following two important characteristics of OO versus structured approach are excerpted from Mohanarajah (2002):

- Higher (Object) Level Abstraction - Object level abstraction is possible in OO approach (OO-A). But in structured approach, only modular level abstraction (and so Abstract Data Type - ADT) is available through top-down design.
- Seamless Transition - In OO-A, all the phases share common vocabulary. That is, everything revolves around object. But, in structured approach, different phases use different tools and strategies during the SW development process.

Object Level Abstraction

The following figure (Figure 1) from Mohanarajah (2002) shows the difference in the design diagrams of OO vs. structured approach for returning a book in a small library system.

Returning a Book, as interactions between objects



Returning a Book, as a hierarchy of modules

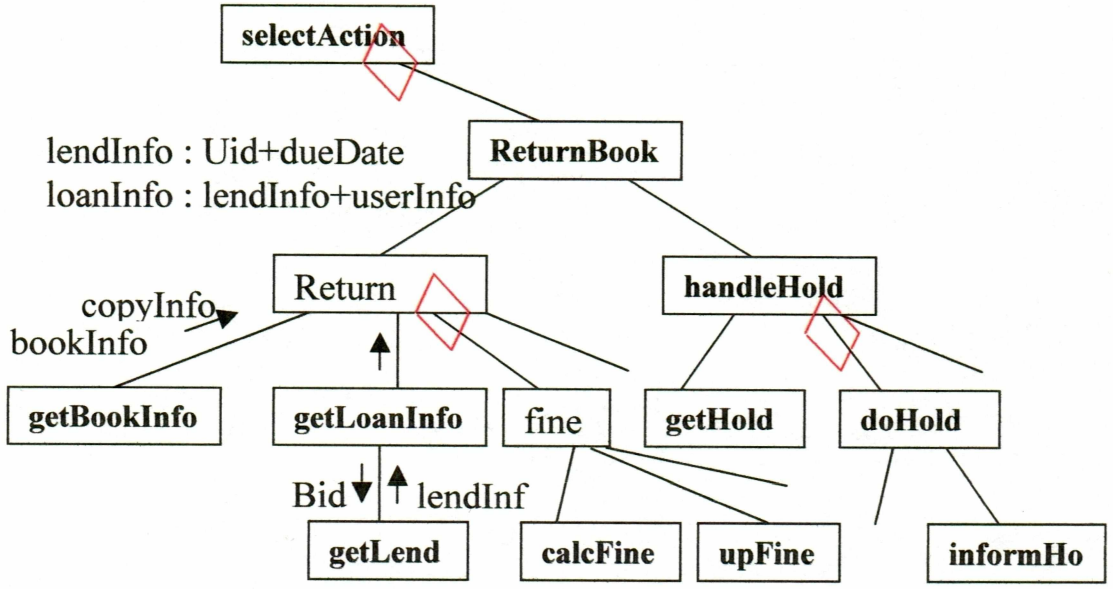


Figure 1: Object Oriented Vs Structured Approach

Seamless Transition

The OO Approach uses the same language - objects for all phases. This can be seen in the following figure (OMG, 1999).

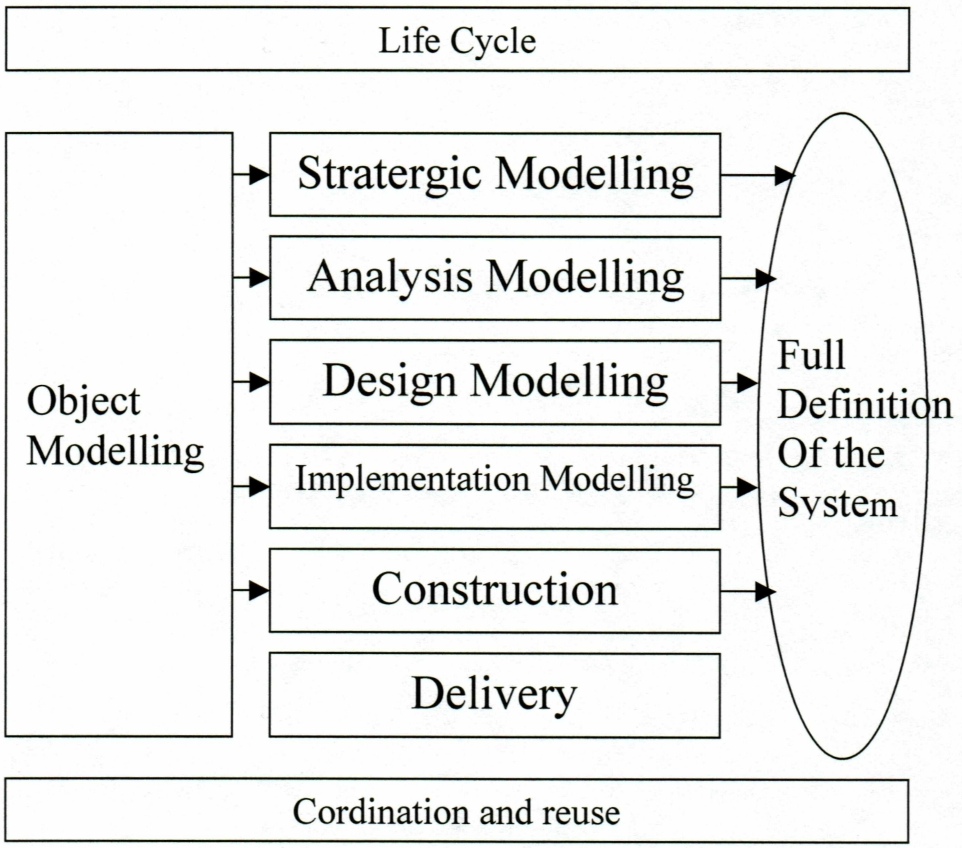


Figure 2: The OO life cycle model proposed by OMG

2.5.2 Paradigms Growth

Structured Development Growth

In the 1990s, while OO was definitely becoming more popular, the structured development technique (SDT) was more widely used by most companies in the software industry. A survey conducted by Survey of Advanced Technology (1991), covering a wide variety of modern software technologies, found the figures given in Table 2:

Table 2: Success Rate with Modern Technologies

Technology	Projects Used on	Success	Effective Penetration
OO/OOPS	3.8%	91.7%	3.5%
Structured Methods	71.4	90.2	64.4
Fourth-generation languages	20.6	86.6	17.8
Relational DBMSs	39.3	84.2	33.1
Model-based systems	23.1	80.7	18.6
End-user computing	25.0	75.0	18.8
Imaging	4.4	70.0	3.1
PC/workstation-based development	27.5	67.8	18.7
Executive information systems	6.9	63.1	4.4
CASE technology	28.8	59.7	17.2
AI/expert systems	3.9	55.3	2.2

As noted by Yourdon (1994) that one is tempted to conclude from these figures that OO is used by a tiny group of enthusiastic, highly successful, fanatics, that is, the very people whose job is to "champion" the technology within the organization.

Object-Oriented Paradigm Growth

By 1998, the OO usage had a tremendous growth in the software technology industry. According to Survey of Advanced Technology (1998) by Chris Pickering that:

- In 1991, less than 20% of Survey respondents were using object-oriented technologies. Today, the figure stands at 54.9%. Java users outnumber the users of all other object-oriented languages combined.
- Component-based development is coming on strong. In 1996, only 5.6% of Survey respondents were using component-based development. In 1998, that figure has grown to 39.2%.

As the economic downturn has had its impact to the entire technology industry, the majority of vendors in the software technology are struggling with new deeper challenge.

The following market impacts on application development are excerpted from Gartner's Prediction 2003: Continued Challenges for Software Industry published on 20 November 2002 (Gartner, 2002):

> The application development (AD) tools market is seeing an increase in hosted development services (HDS). As enterprises emerge from the current economic slump, HDS will be a particularly attractive way of supplementing and augmenting an established or decimated application development organization. The object-oriented

analysis and design (OOAD) tool market is rapidly evolving. Team support and support for component-based and service-oriented architectures (SOA) remain immature and may be a reason for enterprises to avoid long-term commitments (Gartner, 2002).

Based on this prediction, it seems that the usage of the OOAD paradigm will continue to increase in the years to come whereas the component-based technique will slow down its growth in the software technology.

2.5.3 Analysis, Design and Implementation as Levels of Abstraction

The Role of Abstraction

Abstraction is defined as "That which comprises or concentrates in itself the essential qualities of a larger thing or of several things" (Webster's New Collegiate Dictionary, 1958). Abstraction, which is the act of abstracting, is a natural process that allows a person to work with various details for resolving problems that the world presents. The analysis, design and implementation of software systems may be viewed as different levels of abstraction.

Analysis Abstraction

Structured Analysis emphasizes Work Breakdown Structure (WBS) in terms of either functions or procedures. The data model is represented by an Entity-Relationship Diagram (ERD). The process model is represented as a Data Flow Diagram (DFD) that is used to develop an overall functional decomposition. Processes are often specified by use of Structured English as a tool. The data dictionary is an analysis tool that contains definitions of all data flows on the DFD, data stores, and the data elements.

The main focus of Object-Oriented Analysis (OOA) is on identifying the classes and objects that mirror the real-world problem domain (Booch, 1991). During object-oriented analysis, the properties, attributes and methods of the system are emphasized instead of the mechanisms that implement them. OOA provides logical models that are independent of implementation.

Design Abstraction

Structured Design partitions the system into modules and arranges the modules in a hierarchy. The objective of structured design is to have a hierarchical organization of modules that are loosely coupled and highly cohesive. Coupling is a measure of interconnection among modules in a software structure (Pressman, 2001). Cohesion is a measure of the relative functional strength of a module (Pressman, 2001). In software design, it's important to strive for the lowest possible coupling and the highest cohesion (i.e. a module performs only one distinct procedural task) so that the system is easier to understand and less prone to a "ripple effect" (Stevens, Myers and Constantine, 1974) caused when errors occur at one location and propagate through a system. A structure chart (e.g. flow chart) is used to graphically represent a hierarchy of modules and is derived from the DFD developed in the analysis phase. Pseudocode is sometimes used in structured design to clarify the detailed internal logic and/or procedure of some of the black boxes on the structure chart. Pseudocode is an informal and very flexible programming language that is not intended to be executed on a machine but is used to organize a programmer's thoughts prior to coding (Page-Jones, 1988).

System Design is a translation from the analysis, and it produces a physical model that is closer to the software structure. Object-oriented design (OOD) leads to an OO decomposition (Booch, 1991). Booch's method uses Class, Object, Module, and Process Diagrams for OOD. The Class Diagram shows classes and the relationships between the classes. The Object Diagram shows the existence of objects and message passing between those objects. The Module Diagram shows the allocation of classes and objects to modules, and the Process Diagram shows the allocation of processes to physical processors.

Implementation or Programming Abstraction

Implementation or programming, an extension of the design process, is to translate the design into a machine-readable form. Structured programs focus mainly on procedures or algorithms and use controls such as sequence, iteration (e.g., do-while loop), and selection (e.g., if-then or case statement). Procedural languages (also referred to as imperative languages) such as C, Pascal, COBOL, and FORTRAN are highly used for structured programming.

OO programs are organized as cooperative collections of objects that accomplish tasks by passing messages to each other. An OO program would typically use objects as building blocks, where each object would be an instance of some class, and the program would also implement inheritance relationships between classes (Booch, 1991). Languages such as Smalltalk, Simula, C++, Java, Eiffel and Visual Basic, which support OO concepts, are the choices for OO programming.

The points of analysis, design and implementation as levels of abstraction for the traditional (structured) approach vs. the OO approach are summarized as follows (see Table 3):

Table 3: Summary of Levels of Abstractions

Abstraction Level	Structured Approach	Object-Oriented (OO) Approach
Analysis (emphasizes on What need to be done)	Focus on procedures and/or functions	Focus on objects defined with real-world concepts and derived from the problems domain
Design (emphasizes on How to accomplish tasks defined in the analysis)	Focus on functional decomposition and hierarchy of functions	Focus on OO decomposition and Hierarchy of Classes and Objects
Implementation or Programming	Procedures/algorithms (Language options are: C, Pascal, COBOL, and FORTRAN)	Objects and their interactions (message passing); inheritance; algorithms for operations or methods used by objects (Language options are: Smalltalk, Simula, C++, Java, Eiffel and Visual Basic)

2.5.4 Unified Modeling Language (UML)

UML Definition and Background

The Unified Modeling Language (UML) is the industry-standard language for specifying, constructing, visualizing, and documenting the artifacts of a software-

intensive system. It simplifies the complex process of software design - making a "blueprint" for construction (Rational.com, 2002). UML was developed jointly by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational Software Corporation, with contributions from other leading methodologists, software vendors, and many users. In November, 1997, the Object Management Group (OMG) formally accepted UML as the de facto standard for all object development. As Grady Booch says, "You can model 80 percent of most problems by using about 20 percent of the UML" (Booch et al., 1999). Many companies are incorporating the UML as a standard in their development process and products, which includes disciplines such as business modeling, requirements management, analysis & design, programming, and testing. Most of the UML components are supported by a variety of computer-aided software engineering (CASE) software packages, such as Rational Software's Rational Rose, Platinum Technology's Paradigm Plus, Visible System's Visible Analyst, and Microsoft's Visual Studio, etc. Under the OMG's open, vendor-neutral process, the UML continues to evolve to meet changing market needs; revisions are being made to use its expressive power with new emerging technologies such as Java, EJB and XML (OMG, 1999).

Diagramming Techniques

The purpose of UML is to provide a common vocabulary of object-based terms and diagramming techniques that are flexible and consistent enough to model any systems development project throughout the SDLC from analysis to implementation. Page-Jones emphasizes the importance of UML for capturing the structure of object – oriented systems at a level above that of individual lines of code, and it can be expressed

in diagrams that span the gamut of constructs that appear in typical object-oriented systems (Page-Jones, 2000). These diagrams including a set of nine object diagramming techniques used to model a system are defined in Appendix (Dennis and Wixom, 2000). All nine diagramming techniques use the same syntax and notations across all phases of the SDLC, making it easier for analysts and developers to learn the language. Among these diagramming techniques, use case diagrams, sequence diagram, class diagrams, and statechart diagrams have dominated the object-oriented projects.

The key building block of UML is the use case diagramming technique (Dennis and Wixom, 2000). UML requires the analyst and developer to break the system into small use cases and logical pieces of the system and then deal with each piece separately. This approach makes the UML a very good ideal to represent a large and complex system. By contrast, the traditional SDLC approach requires analysts and developers to create DFDs and ERDs that attempt to cover the entire system in one diagram. This DFDs and ERDs approach in traditional SDLC is, however, replaced by the use case, sequence, class and statechart diagramming techniques integrations of UML in the OO paradigm.

CHAPTER 3

RESEARCH on OO MIGRATION PROBLEMS

3.1 Review of the Problem

The problem of moving to OO has become very important. An understanding of potential problems from migrating to the new paradigm helps managers make sound decisions and better anticipate in problem resolutions during the paradigm shift. Some of the difficulties and concerns are as follows:

- Difficulty of changing from a structured to an OO mindset: People who have background and education in the traditional paradigm may find it more difficult to make the paradigm shift than those who are relatively new or inexperienced in software development. As Kuhn (1970) observed that the people who cope with the new paradigm better, in terms of achieving its fundamental inventions, are those who are either very young or very new to the field (Kuhn, 1970). The migration to the OO paradigm from the traditional approach requires changes in organizational processes, policies and procedures that may have impact in work practice and norm. This has some implications as well. Object Orientation has concepts that are quite different from the traditional approach and therefore may involve a steep learning curve.
- Current investments in technology: Organization may be unwilling to make the transition to OO because of enormous investments made in prior software technology. This is a reasonable concern. For example, some companies has invested heavily in

the mainframe (e.g, IBM 3090 or Honeywell 66/40) with older database technologies, such as network and hierarchical database, and they are unwilling to choose the open system implementation with relational database, although its benefits are acknowledged. OO is much more complex and technically incompatible than systems existing in most of organizations; thus moving to OO is a difficult decision to make for many organizations. In addition, even if organizations decide to make the change to OO, the maintenance of existing older systems must be addressed.

- The shift to OO may require major organizational and structural reorientations, as well as changes in relationships, policies, norms, and roles (Orlikowski, 1993). In many organizations, the Information Technology (IT) mission may also have to be redefined, and sometime a reorganization of the whole department is enviable. This may directly impact the reporting structure and staffing of the technical personnel.

While the difficulties and concerns mentioned above are only a small portion of the total ramifications of moving to a new paradigm, they do demonstrate the nature of the impact that adopting OO can have on an organization.

Human beings have a tendency to resist change. The greater the change is, the higher the demands are on them to accommodate the change. People are dynamic. Some are reluctant to acquire new skill sets to learn new methods and techniques, and to change their fundamental practice for the development of software while the others adopt change easily. In order to make a smoother paradigm shift, managers must understand the implications involved in the migration of the new paradigm. The management

implications of moving to OO from the traditional approach are identified and discussed in the next section.

3.2 Management Implications

The discussion of the management implications described in this section has been taken mainly from the works of: (Yourdon, 1994), (Williams, 1995), (Page-Jones, 2001), (Pfeiffer and Zhang, 2002), (Yoder, 1997) and (Shah et al, 1997).

It is important to note that no one method, technology or technique will solve all the problems. There will always be complications, and these complications must be expected and planned for. Naturally, there are risks involved when adopting new technologies that organizations are not familiar with. Many of the risks come from not understanding how to correctly implement the new technology. Lack of understanding and unrealistic expectations of new technologies seem to be the common denominators for the delay of their proper use. A clear view with understanding of these types of problems can help managers and developers to better anticipate the future and make sound decisions when migrating to OO technologies.

Some of the management issues of moving to OOSD are briefly mentioned in the section of 2.4.4 Pros and Cons of OO Approach. The purpose of the rest of this chapter is to describe different types of problems and challenges that manager and/or developers should try to avoid or resolve when migrating to or developing applications in an OOSD environment. Many of the issues described below are common to any software development as they are significant in the context of OOSD. In addition, suggested

approaches to avoid these problems can enable managers or system developers to truly capitalize on the benefits of OOSD.

Problems with Transferring OO Technology

Training Issues

The OO way of thinking is very different from the traditional approach; therefore, it could be difficult to learn for some people. During the early stages of the OO migration, the OO technologies could be misused, abused or processed in an inefficient manner. Lack of experienced and trained managers and/or developers could lead to unrealistic expectations and misunderstanding the nature of the new technologies.

Management Expectation Issues

Due to lack of understanding the new technologies, management might have unrealistic expectation of the new paradigm. For example, managers may expect their developers to learn Smalltalk, the problem domain, and the framework including mapping the OO technology to a relational model within a few months. They may also put a lot of pressure on their people to perform within this time frame without realizing that there is a large learning curve for them to learn the new technology.

Management Commitment Issues

Some managers still believe that there is a high-risk factor for integrating OO technology; therefore, management support sometimes has not been forthcoming and the committed resources have been very limited. For example, the management may give strong support in purchasing the equipment required for the OO technology, but cut back on the training and/or tools costs.

Conceptual Issues

Managers and developers are sometimes confused about what OOSD is and what it entails. Some developers and managers today think that OOSD simply means defining classes, objects, and methods without a true understanding of the nature of the new paradigm. They often assume that using OOSD will eliminate various development bottlenecks since promises are made that OOSD will reduce the management of complexity and will provide architectural modularity. These misconceptions may lead some to believe that OOSD is a magic bullet for any SW project.

Political Issues

Political pitfalls are probably the most deceptive and yet common among managers and users, because they have little to do with skill, technology or the worthwhile nature of the project. These pitfalls have to do with blame, power, image, personalities, favors, tradition, credit and control for the parties involved in an organization. Developers are prone to stumble into political pitfalls because developers like to think of themselves as rational technical support and assume that others will think as they do and be motivated as they are.

Analysis and Design Issues

Developers often carry into OOSD a bias toward the traditional SDLC. Analysis and design issues come from the struggle to manage complexity, while gaining a sense of how object technology works. Two possible situations are related to this issue. First, developers may take a SDLC approach to analysis and design and then attempt to implement it by using object-oriented techniques. Second, developers may attempt to use

an OOAD approach, but then create classes with their hierarchies and connections that are more along the lines of traditional programs. In both cases, there may be a confusion about the relationship between object classes that would result in poor solutions to the problems, loss of benefits of OOSD, and disillusionment with OOSD [Adhikari, 1995].

Environment, Language and Tool Issues

Environments, languages, and tools constitute the most controversial area of OOSD and are most subject to change as new technologies are developed. The environment comprises the operating systems and application environments in which a given project will run: Windows, OS/2, Unix and others. Languages include the various object-oriented languages used to implement object design: Smalltalk, C++, CLOS, and others. Tools are what developers use to create and test the application: editors, compilers, browsers, source code management systems, computer-aided software engineering (CASE) packages and others (Taylor, 1990).

Selection of languages, tools and environment is an important consideration for the integration of the new paradigm. There can be incompatibilities between languages and tools if they are not chosen carefully. Selection of the wrong environment, language, or tool can affect the OOD project with constant problems or bugs in the development of applications.

Implementation Issues

Like many other of the issues mentioned above, the implementation issues also apply to any new software development, not just to OOSD. In fact, it is pandemic to software development and has been addressed repeatedly over the past thirty years

(Lucas, 1989). The last decade of tool development, language refinement, system evolution and hardware advances has given software developers the power to build applications quickly. However, many faults and failings of software development come from neglecting other software engineering activities such as comprehensive analysis, design, development scheduling and deliverables, and proper planning for system testing and installation/conversion (Rabin, 1995).

Rapid prototyping can fool users into believing that completion of that task will be just as fast and easy. The real danger is that coding fast appears to work for a while, but often there is little immediate feedback to engineers that they are on the wrong path. This often leads to the establishment of an inappropriate architecture and feature implementation.

Class and Object Issues

There are many problems in this category. Each one of the following can lead to poor hierarchy and class design, unpredictable behavior of objects, unnecessary complexity in the project, loss of OOSD benefits, low rate of code reuse and product instability (Bosworth, 1992).

- Confusion of is-a, has-a, and is-implemented-using relationships;
- Confusion of interface inheritance with implementation inheritance;
- Use of inheritance to violate encapsulation;
- Use of multiple inheritance (MI) to invert the is-a relationship; and
- Use of multiple inheritances in any circumstances.

These and similar misuses of inheritance seem to have short-term values and work at the time, but they usually come back to haunt developers in the end.

Reuse issues

One of the primary goals of OOSD has been reuse of code. But, it turned out that for first-time projects, OOSD was more difficult and took longer, unless developers could use existing code in class libraries and other tools. So, it seems that reuse is a benefit of OOSD, but that the payoff will be in the long run, not the short term.

The coding for reuse is harder and takes longer because it takes time and effort up front. As Burd and McDermid (Burd and McDermid, 92) note that: "Risks are involved in all software developments, however, often those projects which employ reuse are susceptible to greater risks than those which do not."

Since OOSD is promoted and adopted as a means to get software developed more quickly, the initial design effort is usually neglected, and the result is a lot of work after the fact to make existing software into a reusable shape. Further, management expects reusable software for subsequent projects. This is seldom the case, because similarities among projects are often small and expectations for reuse may outstrip the skills and experience of the developers involved [Hayes, 1996].

3.3 Research Results and Suggestions

There are many potential problems and challenges involved in using OOSD technologies, just like there are in using traditional methodologies for software systems development. These problems however can be avoided and resolved. If OOSD is used properly, the rewards and benefits can be greater than using traditional approaches. The

following section provides suggestion from research results to avoid and/or resolve the problems identified above.

Suggested Solutions for Training Issues

Integrating OO technology from a traditional system development approach definitely deals with many cultural issues and much care needs to be taken to train people with this new paradigm.

Management must recognize the training needs and a learning curve required for learning the new technologies. It is important to teach the developers on OO thinking and assist them with the migration of OO technology.

The focus must also be put on training management in dealing with cultural behaviors while introducing objects and to assist them in dealing with the fear.

Suggested Solutions for Management Expectation Issues

It is very important for managers to manage expectations: give people adequate time for their learning curve, and to go through the transition with little risk, and to allow them for possible mistakes during the early transition to OO technology.

Suggested Solutions for Management Commitment Issues

Once the decision for the integration to the new OO paradigm is made, management must be committed to provide a strong support in all areas such as purchasing equipments and tools, providing adequate training and whatever other needs might be to ensure the success of the integration.

Suggested Solutions for Conceptual Issues

It is important to recognize that OO approach to architecture, design and coding may require, or at least may work best with, different management and scheduling techniques. Upper management, technical management, and developers must all work together by using realistic schedules, with continuous education, and solid engineering techniques to reduce conceptual problems and gain the most benefits from adopting OOSD.

Managers must also be aware that the time spent at the beginning will save time later.

Suggested Solutions for Political Issues

It is important for the developers to realize that organizational politics exist, are significant, and also cannot be ignored. There are, additionally, two separate but related tasks important in preventing political pitfalls. The first task is the education of management about what OOSD entails, which means that developers had better know it themselves, and know it well enough to explain it to non-technical people. The second task is a need for developers to enlist the support of key people: those who can affect budgets and resources, and those who can affect the scope and direction of major projects (Mattison & Sipolt, 1995).

Suggested Solutions for Analysis and Design Issues

The skills of object-oriented analysis, design and programming, cannot be acquired overnight. Thus, before developers start a project, they must decide whether or not to use OOSD. They have to ensure that OOSD is being adopted for the right reasons and have a good understanding of the risks involved.

The OOSD process is an iterative process. OOSD system design and analysis is equivalent to the basic SDLC approach plus high-level strategy decisions. If a project is going to be developed using OOD, then everyone affected should study object-oriented analysis and design. Developers should proceed slowly and look for ways to simplify and generalize; good object-oriented designs tend to resolve to general principles.

Suggested Solutions for Environment, Language and Tool Issues

OOSD developers should consider the learning curve for a language; some languages require a longer learning curve than others. A tool that works for a single developer writing a small stand-alone application will not necessarily scale to production release of more complex applications. To prevent this pitfall, it is important that developers consider technology, compatibility and economic issues before selection of environment, languages and tools. In addition, developers should validate their selection of tools and languages early. They should try out the complete set of tools that they intend to use for production development as early in the system development as possible.

Suggested Solutions for Implementation Issues

To prevent the implementation problem in OOSD, it is important that analysis and design be sufficiently completed before subsequent phase of SDLC (i.e. coding and testing). It is also important to define standards for design, coding, implementation and documentation for each subsystem and class before coding and implementation take place. This seems obvious, it however is easily overlooked in the OOSD environment.

Suggested Solutions for Class and Object Issues

To avoid these pitfalls, developers should set guidelines to create class hierarchies. For each class to be defined, use the descriptions to determine the relationships with existing classes. Additionally, set up the interface and implementation inheritance according to the class design. Also, set guidelines for how methods should be exported and inherited or overridden.

Class and object design is an art and a skill that comes with time, thought, learning, practice and experience. Class and object implementation is a science, based on careful following of canonical forms, coding standards, pre- and post-conditions, and other aspects of software engineering. By combining these factors, developers can create classes that are logical, relevant, subclassable, portable, loosely coupled and most importantly reusable. This pitfall is far easier to avoid than to correct.

Suggested Solutions for Reuse Issues

To focus on reuse of code, rather than the design of OOSD, usually leads to unnecessary delays and slipped schedules. To avoid this pitfall, developers should plan and design reuse before a single line of code is written. It is also well worth the time and resources necessary to track dependencies and interactions, particularly among objects or subsystems that are expected to be reused. Thus, the reusability of objects must be documented in the design of a system as a fact, instead of as a future potential benefit.

3.4 Conclusion for Management Implications

There are many issues, pitfalls and traps involved in using OOSD technologies, just like there are in using traditional methodologies for the application systems

development. Managers and developers should try to avoid and/or resolve these problems when developing applications in an OOSD environment by understanding the nature of the new paradigm and the issues involved.

OOSD requires more discipline, management and training than traditional software development does. Education and experience are keys for the success of any OOSD project. A company in which upper management, technical management, and developers all work together - using realistic schedules, with continuous education, and solid engineering techniques - will likely have fewer problems and gain the most benefits from adopting OOSD.

CHAPTER 4

SUMMARY AND CONCLUSIONS

4.1 Summary

The key to sustaining organization's competitive advantage in the computing technology market lies in its software engineering capabilities. The basic software engineering philosophy is to use process and technology that support an incremental and evolutionary approach to delivering quality solutions. The traditional structured system development approach, which includes SDLC (i.e. analysis, design, coding, testing and Maintenance) as defined by IEEE, was supposed to overcome "software crisis". However, software is still in crisis while the industry demands more complex application systems than ever. Object Orientation (OO) has been used as an emergence of the new paradigm that provides a viable alternative to traditional approach.

Page-Jones' Fundamentals of Object-Oriented Design in UML and Yourdon's Object-Oriented Systems Design are the conceptual basis for this thesis. The Object-Oriented Programming (OOP) concepts have been around since the 1960s. It was, however, not until the 1980s that object-oriented analysis (OOA) and object-oriented design (OOD) started attracting attention. OO is a new paradigm, and it requires new types of knowledge, new specialists, and significant changes in the mindset, an entirely different way of thinking, representing and solving a problem.

While many well-known software development practitioners believe OO approach has distinct advantages over traditional systems developments, OO is also a fundamentally different paradigm from the traditional approach. A summary of the advantages of OO and the main differences between OO and the traditional approach are described below.

Advantages of OO:

- Better management of complexity
- Enhanced reliability and robustness
- Better understanding of problem domain
- Smoother final integration and test
- Reduced cost
- Increased productivity
- Increased reusability
- Increased maintainability
- Increased extensibility

Main difference between OO and structured approach:

- While structured approach emphasizes on functions, OO emphasizes on the objects.
- In OO approach, all the phases share common vocabulary. But, in structured approach, different phases use different tools and strategies during the SW development process

- OO development follows closely the (fast) prototype life cycle while the older structural methodologies follow mostly a waterfall model life cycle.

There are many issues, pitfalls and traps involved in using OOSD technologies, just like there are in using traditional methodologies for the application systems development. However, these problems can be avoided when developing applications in an OOSD environment by understanding and resolving the nature of the problems. If OOSD is used properly, the rewards and benefits can be greater than using traditional approaches.

The Unified Modeling Language (UML) is the industry-standard language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system (Rational.com, 2002). The purpose of UML is to provide a common vocabulary of object-based terms and diagramming techniques that are flexible and consistent enough to model any systems development project throughout the SDLC from analysis to implementation. UML is a very good way to represent a large and complex system. In the OOSD, the integration of UML's use case, sequence, class and statechart diagramming techniques replaces data flow diagrams (DFDs) and ERDs that attempt to cover the entire system in one diagram in the traditional SDLC approach.

The information of the management implications and suggested solutions provided in Chapter 3 is intended to be used as guidelines for the developers and managers in moving to the OO approach. Developers and managers should try to avoid and/or resolve the management implications when developing applications in an OOSD environment by understanding the nature of the new paradigm and the issues involved.

Chris Pickering of Survey of Advanced Technology (1998) reported that a total of 54.9% of Survey respondents were using object-oriented technologies and Java users outnumber the users of all other object-oriented languages combined. It is clear that the OOSD paradigm and technologies are with us to stay and will help us create the application systems. We must continue our pursuit of the enhancement of the tools and our own education in order to gain the maximum benefit from Object-oriented development.

4.2 Discussion

While OOSD offers significant benefits, it also raises many concerns. How much training is required and what is the expense? Other concerns include the maturity of technology, and related tools, lack of standards, and execution speed. Integrating OOSD into the current development methods will require tools that are open and do not include proprietary technology.

Just like the adoption of any new technology, there is a learning curve involved with the adoption of OOSD. Instant and complete submersion in OOSD can be disastrous, whereas carefully planned and scaled adoption of these new technologies can bring out all the positive advantages that they have to offer. Proper education in the OOSD paradigm and technologies must precede any attempt to use OOSD. In the beginning, it should be used for small-scale, non-mission critical applications, so that the organization may receive quick feedback and make necessary adjustments in its usage of these new technologies. In addition, management must understand what it takes for a

successful migration of the new paradigm and provide strong support and commitment to the success of the migration.

It must be emphasized that the findings in this study are very preliminary conclusion. Further research needs to be explored in supporting the conclusions as they might pertain to the practice of using OO in the real commercial software development.

4.3 Implications of the Research

Object-Orientation (OO) appears to be the wave of the century. As both the complexity and the cost of systems rising rapidly, many organizations are looking for better methods to develop and implement the OO paradigm. Organizations in the software industry that adopts OO technologies are increasing dramatically in the recent years. While the tools and techniques of OO are becoming more mature, many organizations jump on the bandwagon fail to recognize the drastic changes required in the education and training of people in using this technology. The problem of moving to OO has become very important. An understanding of potential problems from migrating to the new paradigm helps managers make a smoother paradigm shift. Some of the difficulties and concerns are as follows:

- Difficulty of changing from a structured to an OO mindset: People who have background and education in the traditional paradigm may find it more difficult to make the paradigm shift than those who are relatively new or inexperienced in software development.

- Current investments in technology: Organization may be unwilling to make the transition to OO because of enormous investments made in prior software technology.
- The shift to OO may require major organizational and structural reorientations, as well as changes in relationships, policies, norms, and roles (Orlikowski, 1993). The degree of shift and the extent of adjustment to be made by the organization depend on its current practices.

The new way of thinking in OO has far reaching implications for individuals in the discipline as well as for organizations. Human beings have a tendency to avoid significant change because of the fear of failures, the uncertainty of moving to something new and so on. The difficulties and concerns mentioned above present only a small portion of the total ramifications of moving to a new paradigm. There are other factors that prevent the organization from moving to new changes.

Organizations, which do not hesitate to spend million dollars on the tools required for the new paradigm, should recognize that the key to the successful adoption and implementation of OO lies not in the tools alone. Training and education alone may not be sufficient either. Strong management support and efforts must be committed to change organizational as well as individual thinking and behavior with the new paradigm shift.

4.4 Future Research

The results and findings of this study are explicable to a large extent and consistent with most of the literatures used for this research. However, there is a need to

further validate the findings. For examples, many of the issues described in this study are common to any software development as they are significant in the context of OOD. Future research should be directed toward providing some real life examples of the issues that organizations face while moving toward the OO new technology.

This study can also be extended to cover the intellectual structure of the entire field of software engineering. This can be easily done by including the details of the subject areas already used in this research and leading authors in other areas of software engineering who have not been included in this study. The findings in this research are based solely on citations and publications. Case studies and surveys may be done for further research to determine:

- The reasons that distinguish organizations that have successfully implemented OO from those that have not been successful in making the transition to OO.
- The real problems involved in migrating to OO from the traditional approach.
- The growth rate of OO in the current software industry.
- The impact of OO on an organization.

The migration of the new paradigm needs more attention than they have received in the past. A good understanding of the nature of the software development and implications involved will help manager to better anticipate future paradigm shift. It is highly recommended that an in-depth analysis of the software development approaches should be done with a clear view to developing more effective ways to the migration of a new paradigm.

ACRONYMS

AD:	Application Development
ADT:	Abstract Data Type
AI:	Artificial Intelligence
CASE:	Computer Added Software Engineering
DBMS:	Database Management System
DFD:	Data Flow Diagram
ERD:	Entity-Relationship Diagram
GUI:	Graphic Users Interface
HDS:	Hosted Development Services
IEEE:	Institute of Electrical and Electronics Engineers
IT:	Information Technology
OMG:	Object Management Group
OO:	Object Orientation
OO-A:	Object-Oriented Approach
OOA:	Object-Oriented Analysis
OOD:	Object-Oriented Design
OOP:	Object-Oriented Programming
OOAD:	Object Oriented Analysis and Design
OOSD:	Object Oriented System Development
SADT:	Structured Analysis and Design Technique
SASD:	Structured Analysis and Structured Design

SDLC: Software Development Life Cycle

SDT: Structured Development Technique

SOA: Service-Oriented Architectures

RAD: Rapid Application Development

SW: Software

UML: Unified Modeling Language

WBS: Work Breakdown Structure

APPENDIX

UNIFIED MODELING LANGUAGE DIAGRAMS

APPENDIX: Unified Modeling Language Diagrams

Diagram Name	What Diagram shows	What Diagram is Used to Do	What Diagram is Similar To	Systems Development Life Cycle Phases
Use case diagram	The interaction between external users and the system	Capture business requirements for the system	Context diagram	Use cases drive the entire development process
Class diagram	The static nature of a system at the class level	Illustrate the relationships between classes modeled in the system for a specific use case	Data model	Analysis, design
Object diagram	The static nature of a system at the object level	Illustrate the relationships between objects modeled in the system for a specific use case; used when actual instances of the classes will better communicate the model	Data model	Analysis, design
Sequence diagram	The interaction between classes for a given use case, arranged by time sequence	Model the behavior of classes within a use case	Process model	Analysis, design
Collaboration diagram	The interaction between classes for a given use case, *not* arranged by time sequence	Model the behavior of classes within a use case	Process model	Analysis, design

Statechart diagram	Sequence of states that an object can assume, the events that cause an object to transition from state to state, and significant activities and actions that occur as a result	Examine the behavior of one class within a use case	Analysis, design
Activity diagram	A specific business process, or the dynamics of a group of objects; provides a view of flows and what is going on inside a use case or among several classes	Illustrate the flow of activities in a use case	Analysis, design
Component diagram	The physical components (i.e., exe files, dll files) in a design and where they are located	Illustrate the physical structure of the software design,	Architectural analysis, design, implementation
Deployment diagram	The structure of the run-time system; for example, it can show how physical modules of code are distributed across various hardware platforms	Show the mapping of software to hardware components	Architectural analysis design, implementation

BIBLIOGRAPHY

Adhikari, Richard. "Adopting OO Languages? Check Your Mindset at the Door," Software Magazine, November 1995, pp. 49-59.

Booch, Grady, Object Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc., 1991.

Booch G., Jacobson I., Rumbaugh, J., The Unified Modeling Language Reference Manual. Reading, Mass.: Addison-Wesley, 1999.

Boehm, Barry, W., "Software Engineering," IEEE Transactions on Computers, Vol. c-25, Number 12, December 1976, pp. 1226-1241.

Boehm, B. W., "A Spiral Model of Development and Enhancement," Computer, May 1988, pp. 61-72.

Berard, Edward, V., Essays on Object-Oriented Software Engineering, Volume I, Prentice Hall, Inc., 1993.

Burd, E.L., McDermid, J.A.; "Guiding Reuse with Risk Assessments"; University of York Technical Document YCS 183 (1992); York; 1992

Coad, Peter and Yourdon, Edward, Object-Oriented Analysis, Prentice-Hall, Inc., 1990.

DeMarco, T., Structured Analysis and System Specification, Yourdon Press, New Yourk, 1978.

Dennis, Alan and Wixom, Barbara H., "Systems Analysis And Design", 2000, New York: John Wiley & Sons, Inc.

Dijkstra, E. W., "GO TO Statement Considered Harmful," Communications of the ACM, Vol. 11, Number 5, May 1968, p. 341-346.

Gartner, Inc., Prediction 2003: Continued Challenges for Software Industry by Thomas Topolinski and Joanne Correia, http://www4.gartner.com/Init, 20 November 2002.

Hayes, Frank. "The Reality of Object Reuse," Computer World, May 6, 1996, p. 62.

Institute of Electrical and Electronic Engineers for Software Engineering Standards Collection, Spring 1991 Ed., New York: IEEE, Inc.

Kirkerud, Bjorn, Object Oriented Programming with Simula, Addison-Wesley Publishers Pltd., 1989.

Korson, Tim and McGregor, D. John, "Understanding Object-Oriented: A Unifying Paradigm," Communications of the ACM, Vol. 33, Number 9, September 1990, pp. 40-60.

Kuhn, S. Thomas, The Structure of Scientific Revolutions, 2nd Ed., University of Chicago Press, Chicago, 1970.

Nerur, Sridhar, P., Paradigmatic Issues in Software Development: The Case of Object-Orientation, Ann Arbor, MI: UMI Dissertation Services, 1995.

Lucas, Henry. Managing Information Services, McMillan Publishing Company, New York, NY, 1989.

Mattison, Rob & Micheael J. Sipolt. "An Object Lesson in Management," Datamation, July 1, 1995, pp. 51-55.

Mohanarajah, Selvarajah, Object Oriented Analysis and Design (2002), http://www-ist.massey.ac.nz/smohan/oops.htm, 159.254, 2002.

Object Management Group (OMG), Media Alert: OMG Technology Briefing Program, http://www.omg.org/news/pr99/uml_media_alert.html, Oct. 1999.

Orlikowski, Wanda, J., "Case Tools as Organizational Change: Investigating Incremental and Radical Changes in Systems Development," MIS Quarterly, September 1993, pp. 309-340.

Page-Jones, Meilir, Practical Guide to Structured Systems Design, 2nd Ed., Eglewood Cliffs, N.J.: Prentice-Hall, Inc. 1988.

Page-Jones, Meilir, What Every Programmer Should Know About Object-Oriented Design, New York: Dorset House Publishing, 1995.

Page-Jones, Meilir, Fundamentals of Object-Oriented Design in UML, Eglewood Cliffs, New York: Dorset House Publishing, 2000.

Pei, Daniel, Object Oriented Analysis and Design, Information Systems Management, Winter 95, Vol 12 Issue 1, p54.

Pfeiffer, Simon and Zhang, Yongqin, SENG 613 - Mastering the SW Lifecycle: Structured Analysis / Design, http://www.enel.ucalgary.ca/~pfeiffer/courses/613/group/sasd.htm, January 10, 2002.

Pressman, Roger S., Software Engineering: A Practitioner's Approach, 5th Ed. New York: McGraw-Hill Higher Education, 2001.

Rabin, Steven, "Host Developers to Object Technicians," Information Systems Management, Summer 1995, pp. 30-39.

Rational Software Corporation, Unified Modeling Language, http://www.rational.com/, 2002.

Shah, Vivek, Sivitanides, Marcos and Martin, Roy, Pitfalls of Object-Oriented Development, http://www.westga.edu/~bquest/1997/object.html, 1997.

Schneider, Michael G. and Gersting, Judith L., An Invitation to Computer Science, 2nd Ed., Pacific Grove, CA: Brooks/Cole Publishing Company, 1999.

Stevens, W. P., Myers, G. J. and Constantine, L. L., "Structured Design," IBM Systems Journal, vol. 13, no. 2, May 1974, pp. 115-139.

Sommerville, Ian. Software Engineering, 6th Ed. Pearson Education Limited, 2001.

Survey of Advanced Technology by Chris Pickering, Published by Systems Development, Inc., http://www.cutter.com/itreports/sat98.html, 1998.

Survey of Advanced Technology by Chris Pickering, Published by Systems Development, Inc., http://www.cutter.com/itreports/sat98.html, 1991.

Taylor, David A. Object-Oriented Technology: A Manager's Guide, Addision-Wesely, Reading, MA, 1990.

University of Missouri - St.Louis, "The Ups and Downs of Object-Oriented Systems Development," http://www.umsl.edu/~s1014472/pres/report2.html, 2000.

Ward, P. T. and Mellor, S. J., Structured Development for Real-Time System, Vol. 1, 2, and 3, Yourdon Press, New York, 1985.

Webster's New Collegiate Dictionary, Second edition, Springfield, Mass: G. & C. Merriam Co., Publishers, 1958.

Wegner, Peter, "Concepts and Paradigms of Object-Oriented Programming," OOPS Messenger, l(1), 1990, pp. 8-84.

Williams, John D., What Every Software Manager must know to succeed with Object Technology, New York: SIGS Books, 1995.

Yoder, Joe, Problems with Transferring OO Technology, http://www.joeyoder.com/papers/oopsla96/oopeople/node2.html, August 27, 1997.

Yourdon, Edward, Object-Oriented Systems Design: An Integrated Approach, New Jersey: Prentice Hall, Inc., 1994.

Yourdon, E., Managing the Structured Techniques, New York: Yourdon Inc., 1986.

Yourdon, E. and Constantine, L. L., Structured Design – Fundamentals of a Discipline of Computer Programs and systems Design, Prentice Hall, Englewood Ciffs, New Jersey, 1979.

Zeigler, P. Bernard, Object-Oriented Simulation of Hierarchical, Modular Models of Intelligent Agents, and Endomorphic Systems, Academic Press, Boston, 1990.